APPLICATION

FOR

UNITED STATES LETTERS PATENT

APPENDIX D

TITLE: **PRONUNCIATION GENERATION IN SPEECH RECOGNITION**

APPLICANT: **JAMES K. BAKER, GREGORY J. GADBOIS, CHARLES E. INGOLD, STIJN A. VANEVEN AND JOEL PARK**

Appendix D

```
/*///////////////////////////////////////////////////////////////////////////
//  FILE:           phnspell.cpp
//  CREATED:        2-Jan-96
//  AUTHOR:                 Charles Ingold
//  DESCRIPTION:            Pron spelling and frequency table class.
//
//    Copyright (C) Dragon Systems, 1995-1996
//    DRAGON SYSTEMS CONFIDENTIAL
//
// Revision history log
    VSS revision history.  Do not edit by hand.
    $Log: /PQ/prons/phnspell.cpp $

    3     3/25/97 16:21 Chuck
    PHONEQUERY Ver 0.01.172
    removed old-trec-project only SDWord_{Set,Get}Frequency calls to build
    under MREC.  This means that for now we don't rescale the one-grams in
    the pron guesser.

*////////////////////////////////////////////////////////////////
#include "stdafx.h"
//#include "trec.h"
//#include "myassert.h"
//#include "cutil.h"
//#include "sdapi.h"
//#include "apputil.h"
//#include "ckapi.h"

#if 0
      #ifdef TREC
      extern SDInteger SDWord_GetFrequency( SDhVoc hVoc, SDhWord hWord ) ;
      extern void SDWord_SetFrequency( SDhVoc hVoc, SDhWord hWord, SDInteger
freq ) ;
      #endif

      #define TOTAL_VOC_FREQ 200000000
#endif

#include <ctype.h>

#include "phnspell.h"
//#include "results.h"

GlobalParDef( gParUsePhnSpellLM, "phnspell/UsePhnSpellLM", E_INT, 1,
              "1 means use the frequencies from the PhnSpellArray to build
pron-networks\n"
              "0 means use the same frequency for all words in a given SDState
in pron-networks." ) ;

DEF_ERR( PhnSpell, 1,
             "PhnSpell_ReadAscii() format error line = <%s>" ); // 1 %s line
DEF_ERR( PhnSpell, 2,
             "PhnSpell_ReadAscii() file must be sorted in ascending order on
spelling.\n"
             "     %s < %s\n");
DEF_ERR( PhnSpell, 3,
             "PhnSpell Datafile Version %d unknown.  Try version %d.");
DEF_ERR( PhnSpell, 4,
             "PhnSpell data corrupted.");
DEF_ERR( PhnSpell, 5,
             "Read from disk has failed.");
DEF_ERR( PhnSpell, 6,
                "Write to disk has failed.");
DEF_ERR( PhnSpell, 7,    // 1 %c unknown char 2 %s spelling
                "No known pronunciation for character '%c' in spelling '%s'.");


#define PHN_SPELL_ARRAY_FILE_VERSION 1

#if 0

/////////////////////////////////////////////////////////////////////////
      //
      //  Helper func for getGuessRule, rescales the frequencies for words in
hState
      //  by scaleFactor
      //
      #ifdef TREC
      void rescaleWordFreqs(SDhVoc hVoc, SDhState hState, SDInteger
scaleFactor)
      {
         SDhWordIterator hSWIter =
          SDState_IterateWords(hVoc, hState, SDHCOLL_NOCOLLATION,
SD_WORD_NORESTRICTION,"") ;

         SDInteger freq = 0;
         SDInteger newFreq = 0;
         SDhWord hWord = SDWord_Next(hSWIter) ;

         while ( hWord != 0 )
         {
               freq = SDWord_GetFrequency(hVoc, hWord) ;
               newFreq = freq * scaleFactor;
               SDWord_SetFrequency(hVoc, hWord, newFreq);
               hWord = SDWord_Next(hSWIter) ;
         }
            SDWord_EndIteration(hSWIter);
      }
      #endif
#endif

///////////////////////////////////////////////////////////////
// Populate hGuessState with words based on prons for pPhnSpell.
void PhnSpellArray::getGuessWords(SDhVoc hScratchVoc,
//              SDhState hScratchParentState,
                  SDhState  hGuessState,
                  char *szGuessStateName,
                    SDInteger *pTotalFreq,
                    PhnSpell *pPhnSpell) {

      /// Create SDhWord items for each pron
    PronOffset *pPronOffset = getOffsetPron0( pPhnSpell );
    SDWordLMInfo wordLMInfo;
    wordLMInfo.version = CURRENT_LMINFO_VERSION;
    wordLMInfo.nUnigramCount= 1;
    strncpy( (char *) wordLMInfo.categoryName,  "XX", 3);

      SDInteger bestFreq = 0;
    PronOffset *pBestPronOffset = NULL;

    // Get the set of phonetic string fragments for each
    // substring which matches the string from this position
    while ( pPronOffset ) {
            // force double 0-byte termination
        int lenPronBuf = 2 + strlen( getPron( pPronOffset ) );
        char *pPronBuf = DgnNewArray(char, lenPronBuf);
        memset(pPronBuf, 0, lenPronBuf);
        strncpy(pPronBuf, getPron( pPronOffset ) , lenPronBuf - 2);
//              pPronBuf[ lenPronBuf - 2 ] = ' ';

        SDhWord hWord = 0;
        int wordLen = strlen(pPronBuf) + strlen(szGuessStateName) + 2 ;
        char *pWordName= DgnNewArray(char, wordLen);
        memset(pWordName, 0, wordLen);
        sprintf(pWordName+strlen(pWordName), "%s\\%s", szGuessStateName,
pPronBuf);


            CHK_SDAPI (hWord = SDWord_GetHandle(hScratchVoc, pWordName) );
            if ( hWord == 0 ) {

            SDInteger freq = getFrequency( pPronOffset );
                  if ( freq > bestFreq) {
                        bestFreq = freq;
                        pBestPronOffset= pPronOffset;
                  }
                  if ( (int) gParUsePhnSpellLM == 0 ) {
                        freq = 1;
                  }


/* #ifdef TREC
              CHK_SDAPI( hWord = SDWord_New( hScratchVoc, pPronBuf ) );
              assert(hWord);
              SDWord_SetFrequency( hScratchVoc, hWord, freq);
#else
*/
              wordLMInfo.nUnigramCount = freq;
              CHK_SDAPI( hWord = SDWord_NewWithLMInfo(hScratchVoc,
                                             pWordName, &wordLMInfo) );
// #endif


              pPronBuf[ lenPronBuf - 2 ] = 0;
              CHK_SDAPI( SDWord_SetPronunciations( hScratchVoc, hWord,
                                                (unsigned char*) pPronBuf) );
              pTotalFreq += freq;
              CHK_SDAPI( SDState_AddWord( hScratchVoc, hGuessState, hWord) );
          }
        pPronOffset = getOffsetNextPron(pPronOffset);
        DgnDeleteArray(pPronBuf);
          DgnDeleteArray(pWordName);
      }

#if 0
      #ifdef TREC
    // rescale the frequencies for the words in the state
    SDInteger freqScaleFactor = TOTAL_VOC_FREQ / pTotalFreq;
    rescaleWordFreqs(hScratchVoc, hGuessState, freqScaleFactor);
      #endif
#endif
    //      xDumpState(hScratchVoc, phStateArray[ spellLen ]);

      // Record the best pron based on unigram scores
      CHK_SDAPI( SDhEnv hStateEnv = SDState_AccessEnv( hScratchVoc,
hGuessState, SDENV_ADDTO ) );
      CHK_SDAPI( SDEnv_SetData(hStateEnv, "bestPron",
getPron(pBestPronOffset), strlen(getPron(pBestPronOffset))+1) );
      CHK_SDAPI( SDEnv_SetData(hStateEnv, "bestFreq", &bestFreq,
sizeof(SDInteger)) );
}


// We use alternates for states, rather than words, so
// we need to know the number of different length spelling entries
// which match the unknown word.  For now, we just add unneeded
// start/end alternative operations.

// Fill in phStateArray with states for partial matches on szSpelling
void PhnSpellArray::getGuessStates(SDhVoc hScratchVoc,
                                SDhState hScratchParentState,
                                        SDhState *phStateArray,
                                const char *szSpelling)
{
      const int lenUnknown = strlen(szSpelling);
    char *pStateName = DgnNewArray(char, 2 * (lenUnknown + 1)+ 3 );
    SDInteger *totalFreqArray = DgnNewArray( SDInteger, lenUnknown + 1 );
    memset(totalFreqArray, 0, (lenUnknown+1) * sizeof(SDInteger) );
    int nSpellMatches = 0;
    PhnSpell *pPhnSpell = NULL;

    for (pPhnSpell = firstPhnSpellMatch(szSpelling);
         pPhnSpell != 0;
         pPhnSpell = nextPhnSpellMatch( pPhnSpell ) ) {

        nSpellMatches++;
        int spellLen = wcslen( pPhnSpell );

        // From here, we transition spellLen-many chars further into
        // word, and we had better land on either another char in
        // unknown, or the terminal node
        assert (spellLen <= lenUnknown);

        /// Does state exist? Create state if necessary
            memmove(pStateName, szSpelling, spellLen);
            pStateName[spellLen]= 0;
        CHK_SDAPI( phStateArray[spellLen] = SDState_GetHandle(hScratchVoc,
pStateName,

        hScratchParentState) );
        if (phStateArray[spellLen] == 0) {
            CHK_SDAPI( SDhState hNewState = SDState_New(hScratchVoc,
hScratchParentState) );
            assert( hNewState);
            CHK_SDAPI( SDState_SetLMAllowed(hScratchVoc, hNewState, 1) );
                CHK_SDAPI( SDState_SetName(hScratchVoc,     hNewState,
pStateName) );
                phStateArray[spellLen] = hNewState;
                  getGuessWords(hScratchVoc, /* hScratchParentState ,*/
phStateArray[spellLen],
                                    pStateName, totalFreqArray + spellLen,
pPhnSpell);
            }


    } // End loop over phnSpell matches
    if (pStateName)
        DgnDeleteArray(pStateName);

    if (totalFreqArray)
        DgnDeleteArray(totalFreqArray);

    if (nSpellMatches == 0) {
            errThrow( USE_ERR( PhnSpell, 7), pPhnSpell[0], pPhnSpell );
      }
}

///////////////////////////////////////////////////////////////////
//
// PhnSpellArray::getGuessRule()
//
// Description:
//      Create a state machine to guess a pron for szSpelling.
//
// Return:
//   The SDhRule for the first character in pWord.
//
//   We create an array of RuleItems which looks like:
//
// RULE <string>
//   {StartAlternates}
//          [StartSequence]
//                STATE "prefix"
//      RULE "suffix"
//          [EndSequence] Frequency
// {EndALT}
//
// The <> are mandatory.  We have both {} items or neither of them, and they
// may contain a number of Start/EndSequence items.  We have all [] items or
// none of them.

// How many rule items for each rule?
// Estimate 2+(4*nWords):  StartAlt + nWords*(StartSeq Word Rule EndSeq) +
EndAlt
// since most words are in a sequence which references another rule
// Build a pron network for szSpelling. do this by building a rule for each
position in the unknown str. Each
// of these rules consists of a list of alternates.  Each alternate
// consists of a sequence of phonetic fragment and an optional reference
// to substring of unknown which matches a
// phoneme/spelling fragment in the PhnSpellArray.  Each rule
//
//  FUTURE: deal with unpronounced chars in a more principled way
//
SDhRule PhnSpellArray::getGuessRule(SDhVoc hScratchVoc,
                                    SDhState hScratchParentState,
                                    const char *szRuleName,
                                    const char *szSpelling)
{
    assert(szSpelling);
    SDhRule hRetRule = 0;
    int lenUnknown = strlen(szSpelling);
    assert(lenUnknown > 0);

    char *unknown = DgnNewArray( char, lenUnknown + 1 );
    assert(unknown);
    for (int i = 0; i < lenUnknown; i++) {
        unknown[i] = CAST( char, tolower(szSpelling[i]) );
    }
    unknown[i] = 0;

    SDhRule hNextRule = 0;
    char pRName[100] = "";

    /// Create a RULE for each position in the unknown word,
    /// process from last char to first, so we know the rule handles.
    SDhRule *RulesArray= DgnNewArray( SDhRule, lenUnknown );

    RuleItemArray ruleItemArray;
    SDRuleItem ruleItem;
    memset(&ruleItem, 0, sizeof(SDRuleItem));

    SDhState *hStateArray= DgnNewArray( SDhState, lenUnknown + 1 );

    int strOffset;
    for (strOffset=lenUnknown-1; strOffset >= 0; strOffset--)
    {
            // Build a rule for the remainder of the word (pSubStr).
        char *pSubStr = unknown + strOffset;
        ruleItemArray.removeAll();

        CHK_SDAPI( hNextRule = SDRule_GetHandle(
hScratchVoc,hScratchParentState, pSubStr) );
        if (hNextRule != 0) {
            RulesArray[strOffset] = hNextRule;
            continue;
        }

            // Insert Start Op Alt at beginning of description.
        ruleItem.type= SD_RULE_STARTOPERATION;
            ruleItem.frequency= 0;
        ruleItem.hVoc= hScratchVoc;
        ruleItem.value.operation=SD_RULE_OPERATION_ALTERNATIVE;
        ruleItemArray.add(ruleItem, 0);

        // Create a state and description for each partial match on the
spelling.
        int nSpellMatches = 0;
        memset(hStateArray, 0, (lenUnknown+1) * sizeof(SDhState) );

        getGuessStates(hScratchVoc, hScratchParentState,
                       hStateArray, pSubStr);

            SDInteger bestFreq = 0;
            SDhRule hBestRule =0;
            SDhState hBestState=0;

        for (int spellLen = 0; strOffset + spellLen <= lenUnknown; spellLen++)

        {
                  if (hStateArray[spellLen] == 0) {
                        continue;
                  }

                  SDhState hCurrentState = hStateArray[ spellLen ];
                  SDhRule hCurrentRule = 0;
                  SDInteger stateFreq = 0;
                  SDInteger ruleFreq = 0;
                  CHK_SDAPI( SDhEnv hStateEnv = SDState_AccessEnv(hScratchVoc,
hCurrentState, SDENV_EXISTING) );
                  CHK_SDAPI( SDEnv_GetData(hStateEnv, "bestFreq", &stateFreq,
sizeof(SDInteger)) );

                  // Add a description for the new state
              if (strOffset + spellLen < lenUnknown) {    // We go to another
char in unknown

                         hCurrentRule = RulesArray[strOffset + spellLen];


                  // Add StartOperationSequence item
                  ruleItem.type=SD_RULE_STARTOPERATION;
                  ruleItem.frequency= 0; // pPhnSpell->getFreq();
                  ruleItem.hVoc= hScratchVoc;
                  ruleItem.value.operation=SD_RULE_OPERATION_SEQUENCE;
                  ruleItemArray.add(ruleItem);

                  // Add State item
                  ruleItem.type=SD_RULE_STATE;
                  ruleItem.frequency= 0;
                  ruleItem.hVoc= hScratchVoc;
                  ruleItem.value.hState= hCurrentState;
                  ruleItemArray.add(ruleItem);

                  // Add Rule item for next rule
                  ruleItem.type=SD_RULE_RULE;
                  ruleItem.frequency=0;
                  ruleItem.hVoc= hScratchVoc;
                  ruleItem.value.hRule = hCurrentRule;
                  ruleItemArray.add(ruleItem);

                  // Add EndOperationSequence item
                  ruleItem.type=SD_RULE_ENDOPERATION;
                  ruleItem.frequency=0;
                  ruleItem.hVoc= hScratchVoc;
                  ruleItem.value.operation=SD_RULE_OPERATION_SEQUENCE;
                  ruleItemArray.add(ruleItem);

                        CHK_SDAPI( SDhEnv hRuleEnv =
SDRule_AccessEnv(hScratchVoc, hCurrentRule, SDENV_EXISTING) );

                        CHK_SDAPI( SDEnv_GetData(hRuleEnv, "bestFreq",
&ruleFreq, sizeof(SDInteger)) );

                  } else {  // We goto to terminal node from here

                  ruleItem.type=SD_RULE_STATE;
                  ruleItem.frequency= 0;
                  ruleItem.hVoc= hScratchVoc;
                  ruleItem.value.hState= hCurrentState;
                  ruleItemArray.add(ruleItem);
              }

                  if (stateFreq + ruleFreq > bestFreq) {
                        bestFreq = stateFreq + ruleFreq;
                        hBestRule = hCurrentRule;
                        hBestState = hCurrentState;
                  }
        }
        ruleItem.type=SD_RULE_ENDOPERATION;
          ruleItem.frequency=0;
          ruleItem.hVoc= hScratchVoc;
          ruleItem.value.operation=SD_RULE_OPERATION_ALTERNATIVE;
          ruleItemArray.add(ruleItem);

          // Add the new rule to the voc
          SDRuleItem *pRuleItems = ruleItemArray.getData();
          int nItems = ruleItemArray.count();
          CHK_SDAPI( SDhRule hNewRule= SDRule_New(hScratchVoc,
hScratchParentState) );
          CHK_SDAPI( SDRule_SetDescription(hScratchVoc, hNewRule, pRuleItems,
nItems) );
          CHK_SDAPI( SDRule_SetName(hScratchVoc, hNewRule, unknown + strOffset)
);
            assert(hNewRule);
              RulesArray[strOffset] = hNewRule;
          //  xDumpRule(hScratchVoc, hRule);

              /// Set the bestPron env var for the new rule
              int rulePronLen = 50;
              int statePronLen = 50;
              char *pBestRulePron = DgnNewArray(char, rulePronLen);
              memset(pBestRulePron, 0, rulePronLen);
              char *pBestStatePron = DgnNewArray(char, statePronLen);

              if( hBestRule ) {
                     CHK_SDAPI( SDhEnv hRuleEnv = SDRule_AccessEnv(hScratchVoc,
hBestRule, SDENV_EXISTING) );
                     CHK_SDAPI( SDEnv_GetData(hRuleEnv, "bestPron",
pBestRulePron, rulePronLen) );
              }
              CHK_SDAPI( SDhEnv hStateEnv = SDState_AccessEnv(hScratchVoc,
hBestState, SDENV_EXISTING) );
              CHK_SDAPI( SDEnv_GetData(hStateEnv, "bestPron", pBestStatePron,
statePronLen) );

              // Record the best pron based on unigram scores
              int bestPronLen = rulePronLen + statePronLen + 1 ;
              char* pNewBestPron= DgnNewArray(char, bestPronLen);
              sprintf (pNewBestPron, "%s%s", pBestStatePron, pBestRulePron);

              CHK_SDAPI( SDhEnv hRuleEnv = SDRule_AccessEnv(hScratchVoc,
hNewRule, SDENV_ADDTO) );
              CHK_SDAPI( SDEnv_SetData(hRuleEnv, "bestFreq", &bestFreq,
sizeof(SDInteger)) );
              CHK_SDAPI( SDEnv_SetData(hRuleEnv, "bestPron", pNewBestPron,
strlen(pNewBestPron)+1) );
        } // End loop over chars in unknown word

     if (hStateArray)
         DgnDeleteArray(hStateArray);

     if (unknown)
         DgnDeleteArray(unknown);

     if (RulesArray) {
         hRetRule = RulesArray[0];
         CHK_SDAPI( SDRule_SetName(hScratchVoc, hRetRule, szRuleName) );
              DgnDeleteArray(RulesArray);
       }
       return hRetRule;
```

```
}

//////////////////////////////////////////////////////////////////////////////
//
// Set mpWCTargetSpell to be a Unicode copy of pTargetSpell and
// return a pointer to PhnSpell entry for the first character in pTargetSpell
//
PhnSpell * PhnSpellArray::firstPhnSpellMatch( const char *pTargetSpell ) {

      size_t targetSize = strlen(pTargetSpell) + 1;
      if ( mnWCTargetSpellSize < targetSize ) {
            DgnDeleteArray(mpWCTargetSpell);
            mpWCTargetSpell = DgnNewArray( wchar_t, targetSize );
            memset(mpWCTargetSpell, 0, (targetSize)*sizeof(wchar_t));
            mnWCTargetSpellSize = (uns16) targetSize;
            assert( (uns32)   mnWCTargetSpellSize == (uns32) targetSize);

      }
      mbstowcs(mpWCTargetSpell, pTargetSpell, targetSize);

      // get the offset for the first char
      WideCharOffset wideCharOffset(mpWCTargetSpell[0], 0);

      int pos;
      int retVal = mWCOffsetTable.find(wideCharOffset,WideCharOffsetCmp,
&pos);
      if( retVal != -1 ) {
      PhnSpell *pPhnSpell = mpPhnSpellData + mWCOffsetTable[pos].getOffset();
            return  pPhnSpell;
      }
      return 0;
}

//////////////////////////////////////////////////////////////////////////////
//
//  Find the entry for the first character in pWCTargetSpell, which has to be
//  the first match for the target spelling.
//
PhnSpell *PhnSpellArray::firstPhnSpellMatch( wchar_t *pWCTargetSpell) {

      size_t targetLen = wcslen(pWCTargetSpell);
      wcscpy(mpWCTargetSpell, pWCTargetSpell);     // BUG check length

   // get the offset for the first char
      WideCharOffset wideCharOffset(mpWCTargetSpell[0], 0);
   int pos;
      int retVal = mWCOffsetTable.find(wideCharOffset,WideCharOffsetCmp,
&pos);

      if( retVal != -1 ) {
      PhnSpell *pPhnSpell = mpPhnSpellData + mWCOffsetTable[pos].getOffset();
            return pPhnSpell;
      }
      return 0;
}

//////////////////////////////////////////////////////////////////////////////
//
// Return the next offset in mpPhnSpellData which is the start of a
// spelling
//
PhnSpell * PhnSpellArray::nextSpelling(PhnSpell *pPhnSpell ) {
   assert( ckPhnSpellPtr( pPhnSpell ) );

      while (*pPhnSpell != PHNSPELL_END_OF_ENTRY) {
            pPhnSpell++;
      }
      pPhnSpell++;
      if(*pPhnSpell == PHNSPELL_END_OF_ENTRY)  // Check for end of PhnSpell
data
            return 0;
      assert( ckPhnSpellPtr( pPhnSpell ) );
      return pPhnSpell;
}


///////////////////////////////////////////////////////////////////////////
//
//  Find the next entry in mpPhnSpellData which is a partial match to
//  mpWCTargetSpell, if there is one.
//
PhnSpell *PhnSpellArray::nextPhnSpellMatch(PhnSpell *pPhnSpell ) {

      assert(mpWCTargetSpell);
      assert( ckPhnSpellPtr( pPhnSpell ) );

      PhnSpell *pNextPhnSpell = pPhnSpell;
      int cmpVal = 0;

      for(;;) {
            pNextPhnSpell = nextSpelling( pNextPhnSpell );
            if ( pNextPhnSpell == 0 )
                  return NULL;
            cmpVal = wcsncmp(mpWCTargetSpell, pNextPhnSpell,
wcslen(pNextPhnSpell));

            if (cmpVal == 0)
                  return pNextPhnSpell;

      // Note that we don't stop until pSpell is > mpWCTargetSpell
      // because if we are looking for "at", we may find "al" first
            if (cmpVal < 0)
         return NULL;
      }
}


///////////////////////////////////////////////////////////////////////////
//
// Get the frequency which is in the uns16 following PhnSpell *,
//
inline SDInteger PhnSpellArray::getFrequency(PhnSpell *pPhnSpell )      {
   assert( ckPhnSpellPtr( pPhnSpell ) );

      return (uns16) *(pPhnSpell +1);
}


///////////////////////////////////////////////////////////////////////////
//
//  Get the offset for the first pron for the spelling at phnSpellOffset
//
PronOffset *PhnSpellArray::getOffsetPron0( PhnSpell *pPhnSpell ) {
	assert( ckPhnSpellPtr( pPhnSpell ) );

	while(*pPhnSpell) {	// advance to the NULL-terminator for the
spelling
	pPhnSpell++;
	}
	pPhnSpell++;		// advance to the pron offset
	assert( *pPhnSpell != PHNSPELL_END_OF_ENTRY );

	assert( ckPhnSpellPtr( pPhnSpell ) );

	return pPhnSpell;
}


///////////////////////////////////////////////////////////////////////////
//
// Return the offset into mpPronData which is in the uns16
// Note that we don't change phnSpellOffset, because getOffsetNextPron()
// expects to skip over a frequency
//
PronOffset *PhnSpellArray::getOffsetNextPron( PronOffset *pPronOffset )
{
	assert( ckPronOffsetPtr( pPronOffset ) );
	pPronOffset++; // advance to frequency
	pPronOffset++; // advance to next pron offset
	assert( ckPronOffsetPtr( pPronOffset ) );

	if(*pPronOffset == PHNSPELL_END_OF_ENTRY)
		return 0;
	return pPronOffset;
}


///////////////////////////////////////////////////////////////////////////
//
// Return a pointer to the pron located at the offset in mpPronData which
// we find at the offset in mpPhnSpellData given by phnSpellOffset.
//
char *PhnSpellArray::getPron(PronOffset *pPronOffset )
{
	assert( ckPronOffsetPtr( pPronOffset ) );

	if ( *pPronOffset == PHNSPELL_END_OF_ENTRY)
		return 0;

	char *pRet = mpPronData + (*pPronOffset);
	return pRet;
}


///////////////////////////////////////////////////////////////////////////
//
//  Writes mpPronData,  mpPhnSpellData, and mWCOffsetTable into a binary file
//  for persistant storage and access by readBinaryFile().
//
void PhnSpellArray::writeBinaryFile(FILE *pOutFile) {

	uns16 fileVersion = PHN_SPELL_ARRAY_FILE_VERSION;
	// write the file format version.
	if (1 != fwrite(&fileVersion, sizeof(uns16), 1, pOutFile) ) {
		errThrow(USE_ERR(PhnSpell, 6));
	}

	if (1 != fwrite(&mnEntries, sizeof(uns16), 1, pOutFile) ) {
		errThrow(USE_ERR(PhnSpell, 6));
	}

	if (1 != fwrite(&mTotalFrequency, sizeof(uns32), 1, pOutFile) ) {
		errThrow(USE_ERR(PhnSpell, 6));
	}

	// write the size of the PhnSpellData
	if (1 != fwrite(&mPhnSpellDataSize, sizeof(PhnSpellOffset), 1, pOutFile)
) {
		errThrow(USE_ERR(PhnSpell, 6));
	}

	// write the PhnSpellData
	if( mPhnSpellDataSize !=
			fwrite(mpPhnSpellData, sizeof(uns16), mPhnSpellDataSize,
pOutFile) ) {
		errThrow(USE_ERR(PhnSpell, 6));
	}

	// write the size of the PronData
	if (1 != fwrite(&mnPronDataSize, sizeof(PronOffset), 1, pOutFile) ) {
		errThrow(USE_ERR(PhnSpell, 6));
	}

	// write the PronData
	if (mnPronDataSize !=
			fwrite(mpPronData, sizeof(char), mnPronDataSize, pOutFile) )
{
		errThrow(USE_ERR(PhnSpell, 6));
	}

	// write the number of elements in the WideCharOffsetTable
	int32 nWCOffsets = mWCOffsetTable.count();
	if (1 != fwrite(&nWCOffsets, sizeof(int32), 1, pOutFile) ) {
		errThrow(USE_ERR(PhnSpell, 6));
	}


	// write the WideCharOffsetTable
	void *pEntry = mWCOffsetTable.first();
	if (nWCOffsets !=
			CAST(int32, fwrite(pEntry, sizeof(WideCharOffset),
nWCOffsets, pOutFile)) ) {
		errThrow(USE_ERR(PhnSpell, 6));
	}
}


/////////////////////////////////////////////////////////////////////////////
//
//  Reads a binary file written by writeBinaryFile() into mpPronData,
//  mpPhnSpellData, and mWCOffsetTable.
//
void PhnSpellArray::readBinaryFile(FILE *pInFile) {

	uns16 fileVersion;
	// read the file format version.
	if (1 != fread(&fileVersion, sizeof(uns16), 1, pInFile) ) {
```

```
            errThrow(USE_ERR(PhnSpell, 5));
        }

        if (fileVersion != PHN_SPELL_ARRAY_FILE_VERSION) {
            errThrow(USE_ERR(PhnSpell, 3), fileVersion,
PHN_SPELL_ARRAY_FILE_VERSION);
        }

        if (1 != fread(&mnEntries, sizeof(uns16), 1, pInFile) ) {
            errThrow(USE_ERR(PhnSpell, 5));
        }

        if (1 != fread(&mTotalFrequency, sizeof(uns32), 1, pInFile) ) {
            errThrow(USE_ERR(PhnSpell, 5));
        }

        // read the size of the PhnSpellData
        if (1 != fread(&mPhnSpellDataSize, sizeof(PhnSpellOffset), 1, pInFile) )
{
            errThrow(USE_ERR(PhnSpell, 5));
        }

        // read the PhnSpellData
        mpPhnSpellData = DgnNewArray( uns16, mPhnSpellDataSize );
        if (mPhnSpellDataSize !=
            fread(mpPhnSpellData, sizeof(uns16), mPhnSpellDataSize, pInFile) )
{
            errThrow(USE_ERR(PhnSpell, 5));
        }


        // read the size of the PronData
        if (1 != fread(&mnPronDataSize, sizeof(PronOffset), 1, pInFile) ) {
            errThrow(USE_ERR(PhnSpell, 5));
        }

        // read the PronData
        mpPronData = DgnNewArray( char, mnPronDataSize );
        if (mnPronDataSize != fread(mpPronData, sizeof(char), mnPronDataSize,
pInFile) ) {
            errThrow(USE_ERR(PhnSpell, 5));
        }


        // read the number of elements in the WideCharOffsetTable
        int32 nWCOffsets;
        if (1 != fread(&nWCOffsets, sizeof(int32), 1, pInFile) ) {
            errThrow(USE_ERR(PhnSpell, 5));
        }

        mWCOffsetTable.setSize(nWCOffsets);

        // read the WideCharOffsetTable
    // Future: find & use READ_MANY (trecutil)
        if (nWCOffsets !=
                    CAST(int32, fread(mWCOffsetTable.getData(),
sizeof(WideCharOffset), nWCOffsets, pInFile)) ) {
            errThrow(USE_ERR(PhnSpell, 5));
        }

        // verifyData();
}
```

```
#if 0
int PhnSpellArray::verifyData()
{
      // check that mpPhnSpellData is in incr alpha order on spellings

      // check (0 < pron offsets <= PronTable size)
      // check mWCOffsetTable offsets
      errThrow( USE_ERR( PhnSpell, 4) );
}
#endif


/////////////////////////////////////////////////////////////////////////////
//
// Compare func for use with mWCOffsetTable which is a DgnAC<>
//
int WideCharOffsetCmp(const void *given, const void *test) {
    assert(given);
    assert(test);

      int result =  wcsncmp( ((WideCharOffset *)given)->getChar(),
                                     ((WideCharOffset *)test)->getChar(),
                                            1 );
      return result;
}



/////////////////////////////////////////////////////////////////////////////
//
// Compare func for use with PronOffsetTbl which is a DgnAC<>
//
int PronOffsetEntryCmp(const void *given, const void *test)
{
    assert(given);
    assert(test);

      int result = strcmp( ( (PronOffsetEntry *) given)->mpStr,
                                     ( (PronOffsetEntry *) test)->mpStr);
      return result;
}

const uns16 DATA_GROWTH_SIZE = 1024;

/////////////////////////////////////////////////////////////////////////////
//
//  looks up pData, which is a pron, in PronOffsetTbl.  If it is found, then
//  the offset for the pron is in the PronOffsetEntry, otherwise we add an
entry
//  to both mpPronData, growing it if necessary, and also add another
PronOffsetEntry to
//  PronOffsetTbl.
//
//  Note that PronOffsetTbl is only around while we are in readAscii().
//
PronOffset PronOffsetTbl::getOffset(char *pData) {

    assert(pData);

    int searchResult;

      // See if we need to grow the DataBlock
    size_t dataLen =  strlen(pData)+1;
      if (mCurrentOffset + dataLen > mnDataSize) {
            char *pNewBuf = DgnNewArray(char, mnDataSize + DATA_GROWTH_SIZE );
            memcpy(pNewBuf, mpDataBlock, mnDataSize);
            DgnDeleteArray(mpDataBlock);
            mpDataBlock = pNewBuf;
            mnDataSize += DATA_GROWTH_SIZE;
      }

    // Figure out offset of data in mpDataBlock
      strcpy(mpDataBlock + mCurrentOffset, pData);
      PronOffsetEntry *pPronOffsetEntry = DgnNew( PronOffsetEntry );
      pPronOffsetEntry->init(mpDataBlock, mCurrentOffset);
    int posData;
      searchResult = find(pPronOffsetEntry, PronOffsetEntryCmp, &posData);

    // Have we seen pData before?
    if (searchResult == -1) {
            // No, update mCurrentOffset and pronOffsetArray.
            uns32 newOffset = mCurrentOffset + dataLen;
            assert (newOffset < 0xffff);
            mCurrentOffset = (PronOffset) newOffset; // char after null.
            add(pPronOffsetEntry, posData);

    } else {
            // Yes, so we have the offset and don't need the OffsetEntry
            memset(mpDataBlock + mCurrentOffset, 0, dataLen);
            DgnDelete(pPronOffsetEntry);
            pPronOffsetEntry = (*this)[searchResult];
      }
      return pPronOffsetEntry->mOffset;
}


////////////////////////////////////////////////////////////////////////////
//
//  build our data structures in memory for mpPronData, mpPhnSpellData, and
//  mWCOffsetTable from an Ascii file of spellings, prons and frequencies.
//
// For now, ASCII file must be in ascending alphabetic order on the spelling.
void PhnSpellArray::readAscii(FILE * pDataFile) {

    assert(pDataFile);

      PronOffsetTbl *pPronTable = DgnNew(PronOffsetTbl);

      PhnSpellOffset nDataBlockSize = 2048;
      uns16 *pDataBlock = DgnNew( uns16[ nDataBlockSize ]) ;
      memset(pDataBlock, 0, nDataBlockSize);
      pDataBlock[0] = PHNSPELL_END_OF_ENTRY;
      uns16 *pCurPos = pDataBlock + 1;

      char linebuf[512] = "";
      char phonemes[128] = "";
      char spelling[128] = "";
      char prevSpelling[128] = "";
      unsigned int frequency = 0;
      int numfields;
      for(;;)
      {
      PhnSpellOffset curOffset = CAST(PhnSpellOffset, pCurPos - pDataBlock);
            // do we need to grow the data block?
```

```
        // In this iteration, we may add as many bytes as
        //   2*strlen(spelling) + 2 bytes for 0-terminator
        // + 2 for pronOffset + 2 for frequency + 2 for entry terminator.
        // so as long as strlen(spelling) is less than 40, we're fine...
        // As of 8/1/96, the longest spelling was bearly into double-digits.
            if (curOffset + 100 > nDataBlockSize)
            {

                PhnSpellOffset newSize = (uns16) (nDataBlockSize +
DATA_GROWTH_SIZE);
                assert( (uns32) newSize == (uns32) (nDataBlockSize +
DATA_GROWTH_SIZE));
                uns16 *pNewBlock = DgnNew(uns16[ newSize ]);
                memcpy( pNewBlock, pDataBlock, curOffset*sizeof(uns16) );
                nDataBlockSize = newSize;
                DgnDeleteArray(pDataBlock);
                pDataBlock = pNewBlock;
                pCurPos = pDataBlock + curOffset;
            }

            if (fgets(linebuf, sizeof(linebuf),pDataFile) == NULL)
            {
                *pCurPos = PHNSPELL_END_OF_ENTRY;
                break;
            }

            numfields = sscanf(linebuf, "%s %s %ud", spelling, phonemes,
&frequency);

        assert(strlen(spelling) < 40);
            if ( numfields < 3  || frequency > 0xffff) {
                errThrow(USE_ERR( PhnSpell, 1), linebuf);
            }

            if (frequency == 0)  // BUG KLUDGE We must not have entries in
table with freq == 0
          frequency = 3;       //  because a word w/ freq == 0 would cause
log(0) eventually

        // Force Ascii file to be in incr alpha order on spelling.
            if (strcmp(spelling, prevSpelling) < 0) {
                errThrow(USE_ERR(PhnSpell, 2), spelling, prevSpelling);
            }

        // Check for new spelling.
            if ( strcmp(spelling, prevSpelling) ) {

          // finish prev. pron/LM list
                *pCurPos = PHNSPELL_END_OF_ENTRY;
                pCurPos++;

                // convert to wide-char string
                size_t targetLen = strlen(spelling);
                wchar_t *pWCSpell = DgnNew( wchar_t[ targetLen + 1 ] );
                memset(pWCSpell, 0, (targetLen+1)*sizeof(wchar_t));
                mbstowcs(pWCSpell, spelling, targetLen);
                wcscpy(UNS16PTOWCP(pCurPos), pWCSpell);   // start new
spelling

                if ( spelling[0] != prevSpelling[0] ) {// enter in
WideCharOffsetTable
                    uns32 wcOffset = pCurPos - pDataBlock;
```

```
                              assert(wcOffset < 0xffff);
                        //    xprintf("%10s %10s %5d %7d\n", spelling, phonemes,
frequency, wcOffset);
                              WideCharOffset wideCharOffset(pWCSpell[0], (uns16)
wcOffset);
                              mWCOffsetTable.add(wideCharOffset);
                        }
                        pCurPos += targetLen + 1;
                        strcpy(prevSpelling, spelling);
                        DgnDeleteArray(pWCSpell);
                  }

                  // write another pron/LM entry
                  PronOffset pronOffset = pPronTable->getOffset(phonemes);
                  *pCurPos = pronOffset;
                  pCurPos++;
                  *pCurPos = (PronOffset) frequency;
                  assert(*pCurPos == frequency);
                  pCurPos++;

                  mTotalFrequency += frequency;
                  mnEntries++;
                  assert((pCurPos - pDataBlock) < nDataBlockSize);
            }
            pCurPos++;
            *pCurPos = PHNSPELL_END_OF_ENTRY;

            mpPronData =  pPronTable->getCopyOfDataBlock();
            mnPronDataSize = pPronTable->getCurrentOffset();
            DgnDelete(pPronTable);
            xprintf("uns16 PhnSpellData[] used %d of %d allocated entries.\n",
                        pCurPos - pDataBlock, nDataBlockSize);


            mPhnSpellDataSize = (uns16) (pCurPos - pDataBlock);
            assert((uns16) mPhnSpellDataSize == (uns16) (pCurPos - pDataBlock));
            mpPhnSpellData = DgnNew( uns16[ mPhnSpellDataSize ] );
            memcpy( mpPhnSpellData, pDataBlock, mPhnSpellDataSize*sizeof(uns16) );
            DgnDeleteArray(pDataBlock);
}


/*////// Old History ///////////////////////////////

    2     3/24/97 5:04p Joel
    PHONEQUERY Ver 0.01.167
     fixes for Chuck's build for TREC okay, but we still don't use
     SDWord_SetFrequency, etc., which does not exist in TREC...
     so the TREC build currently is broken... Chuck can you fix this?

    1     3/24/97 16:30 Chuck
    PHONEQUERY Ver 0.01.165
    Added prons lib

    11    11/07/96 12:11 Chuck
    TAHITI Ver 0.04.390
    Now support Silence in pron guessing, if it shows up in SDResult.
    Fixed ErrThrow for unknown character in spelling.

    10    10/07/96 11:39 Chuck
    TAHITI Ver 0.04.337
```

```
Added error throw for unknown char in spelling.

9     9/30/96 6:36p Joel
TAHITI Ver 0.04.317

8     9/13/96 9:41a Chuck
TAHITI Ver 0.04.307
Changed word fragment names to avoid collisions with real
words (6, a, ...)

7     8/06/96 6:24p Chuck
TAHITI Ver 0.04.252
 We now use pointers instead of offsets when reading PhnSpellArray
 Less work, easier to read, and we have assertions too.
 updated .ans to reflect bug fix in reading frequencies for
pron-networks

6     7/29/96 10:09a Joel
TAHITI Ver 0.04.232

5     7/17/96 2:32p Chuck
TAHITI Ver 0.04.210
 Removed data members used for bookkeeping purposes, that stuff belongs
 to
 the caller now, which is in phnguess.{h,cpp}.

4     7/10/96 3:39p Chuck
TAHITI Ver 0.04.198
 ruleItem array is now DgnAC< SDRuleItem > and much better for it.

3     7/09/96 11:25a Chuck
TAHITI Ver 0.04.197
 avoid compiler bug in getGuessRule.

2     7/08/96 8:10p Chuck
TAHITI Ver 0.04.194

1     5/18/96 11:01p Tim
Moving over from TLIB.
$NoKeywords: $

 Old TLIB revision history follows.
 *tlib-revision-history*
1 phnspell.cpp 02-Feb-96,18:00:50,'CHUCK' TAHITI Ver 0.03.222
2 phnspell.cpp 14-Mar-96,11:12:42,'CHUCK' TAHITI Ver 0.03.321
3 phnspell.cpp 27-Mar-96,10:47:00,'CHUCK' TAHITI Ver 0.03.350
4 phnspell.cpp 01-Apr-96,12:56:44,'CHUCK' TAHITI Ver 0.03.363
5 phnspell.cpp 08-Apr-96,09:18:10,'CHUCK' TAHITI Ver 0.03.375
6 phnspell.cpp 10-Apr-96,18:56:54,'STAN' TAHITI Ver 0.03.382
7 PHNSPELL.CPP 17-Apr-96,09:30:08,'BENT' TAHITI Ver 0.03.402
8 phnspell.cpp 17-Apr-96,14:33:26,'HUGH' TAHITI Ver 0.03.404
9 phnspell.cpp 23-Apr-96,18:04:58,'STAN' TAHITI Ver 0.04.020
10 phnspell.cpp 23-Apr-96,19:40:54,'STAN' TAHITI Ver 0.04.021
11 phnspell.cpp 25-Apr-96,13:19:14,'STAN' TAHITI Ver 0.04.028
12 phnspell.cpp 17-May-96,20:03:48,'CHUCK' TAHITI Ver 0.04.097
13 PHNSPELL.CPP 18-May-96,18:54:52,'TIM' TAHITI Ver 0.04.100
 *tlib-revision-history*

 Revision 13 on Sat May 18 18:54:36 1996 by tim TAHITI Ver 0.04.100

 Revision 12 on Fri May 17 20:03:46 1996 by Chuck TAHITI Ver 0.04.097
   Redesigning interface...
```

```
        Revision 11 on Thu Apr 25 13:19:11 1996 by stan TAHITI Ver 0.04.028
          Backed off get-printf-out-of-low-level changes.

        Revision 10 on Tue Apr 23 19:40:52 1996 by stan TAHITI Ver 0.04.021

        Revision 9 on Tue Apr 23 18:04:54 1996 by stan TAHITI Ver 0.04.020

        Revision 8 on Wed Apr 17 14:33:27 1996 by Hugh TAHITI Ver 0.03.404
        Project version of SDState_IterateWords() is now used

        Revision 7 on Wed Apr 17 09:30:03 1996 by Bent TAHITI Ver 0.03.402

        Revision 6 on Wed Apr 10 18:56:50 1996 by stan TAHITI Ver 0.03.382

        Revision 5 on Mon Apr 08 09:18:08 1996 by Chuck TAHITI Ver 0.03.375
          Support for persistant PhnSpellArray object.
          Added errThrow(s) to code.
          Added read/write binary file functions.

        Revision 4 on Mon Apr 01 12:56:42 1996 by Chuck TAHITI Ver 0.03.363
          Support for gudtest and instrumentation for built/dict words

        Revision 3 on Wed Mar 27 10:46:58 1996 by Chuck TAHITI Ver 0.03.350


        Revision 2 on Thu Mar 14 11:12:40 1996 by Chuck TAHITI Ver 0.03.321

        Revision 1 on Fri Feb 02 18:00:48 1996 by Chuck TAHITI Ver 0.03.222
   //
   ////////////////////////////////////////////////////////////////////////////
   */
```